Relación nº 3

1.- Supongamos que tenemos un lenguaje de programación que tiene el tipo de datos Pila pero no el tipo de datos ARRAY. Un array se caracteriza por ser una lista secuencial de datos a los que se puede acceder de forma directa especificando el índice asociado. Las operaciones fundamentales son:

- Dimensión: establece el tamaño del array, el tipo Indice y el tipo Base.
- Asignación de un dato a una localización particular indicada con un índice.
- Acceso a un dato de una localización determinada del array, indicada con un índice

Consideremos las siguientes definiciones para estas operaciones:

Tipos

```
Array = REGISTRO
            pila:TipoPila
            inf,sup:Z
```

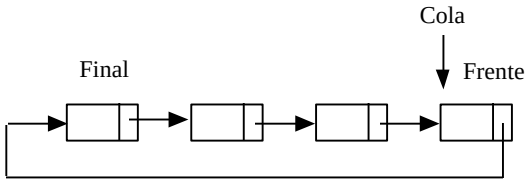
Algoritmo Dimension(VAR a:Array;i,s:Z;inic:TipoElem)
Algoritmo Asignar(VAR a:Array;i:Z;x:TipoElem)
Algoritmo Acceso(a:Array;i:Z):TipoElem

Implementa estas operaciones utilizando el TAD Pila.

2.- Supongamos que una Cola va a representarse mediante una estructura dinámica circular utilizando las siguientes declaraciones:

Tipos

```
PtrCola = PUNTERO A Nodo
Nodo = REGISTRO
            valor:TipoElem
            sig:PtrCola
TipoCola = PtrCola
```



Cola apunta al dato del Frente de la misma, y Cola^.sig apunta al Final.
Con esta representación implementa todas las operaciones del TAD Cola.

3.- A cada implementación del TAD Pila y del TAD Cola realizadas en clase, añade las siguientes operaciones:

Algoritmo TopePila(pila:TipoPila):TipoElem
(* Devuelve el valor que está accesible en la pila sin modificarla *)

Algoritmo FrenteCola(cola:TipoCola):TipoElem
(* Devuelve el valor que está en el Frente de la cola sin modificarla *)

Algoritmo FinalCola(cola:TipoCola):TipoElem
(* Devuelve el valor que está en el Final de la cola sin modificarla *)

4.- Al TAD Lista explicado en clase añade las siguientes operaciones. Impleméntalas utilizando variables dinámicas, y por otro lado también usando variables estáticas.

Algoritmo Longitud(lista:TipoLista):N
Algoritmo EnLista(lista:TipoLista;dato:TipoElem):B
(*Devuelve TRUE si dato está en lista, sin modificarla*)
Algoritmo InsertaDesp(VAR lista:TipoLista;dato_ant,nuevo_dato:TipoElem)
(*Inserta nuevo_dato en la lista después de dato_ant*)
Algoritmo BorraDesp(VAR lista:TipoLista;dato:TipoElem)
(*Borra el elemento que sigue a dato en lista*)

5.- Supongamos que tenemos una máquina con un solo registro y seis instrucciones:

LD	A	Sitúa el operando A en el registro
ST	A	Sitúa el contenido del registro en A
AD	A	Suma A al contenido del registro
SB	A	Resta A del contenido del registro
ML	A	Multiplica el contenido del registro por A
DV	A	Divide el contenido del registro por A

Escribe un programa que acepte una expresión en postfija que contenga operandos simples (formados por letras mayúsculas) y operadores (+,-,*,/), y que escriba en la pantalla la secuencia de instrucciones del tipo anterior para evaluar la expresión en dicha máquina, dejando el resultado en el registro.Por ejemplo, dada la expresión: ABC*+DE-/, el programa debería dar como resultado:

```
LD   B
ML   C
ST   Temp1
LD   A
AD   Temp1
ST   Temp2
LD   D
SB   E
ST   Temp3
LD   Temp2
DV   Temp3
ST   Temp4
LD   Temp4
```

6.- Escribe un programa que lea expresiones que contengan paréntesis ( ), corchetes [ ], y llaves {}, y que compruebe que están perfectamente anidadas.

Puedes suponer que cada línea contiene exactamente un expresión.
Ejemplo:
A[I+X{Y-2}]
LEGAL
B+[3-{X/2})*19+2/(X-7)
ERROR, SE ESPERABA]

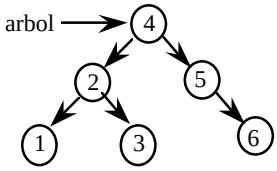
7.- Se entiende por rotación hacia adelante de un paso, que la cabeza de una lista se saque de ésta y se inserte como un nuevo dato al final. La rotación hacia atrás es la inversa de la anterior:

Añade las siguientes operaciones al TAD Lista. Impleméntalas con variables dinámicas.

a) Adelante1, Atrás1 que roten la lista en un dato, en la dirección indicada.
b) Cambia las definiciones para rotar en k datos.

8.- Una cola de doble entrada es una colección de elementos homogéneos en la que se permiten inserciones y extracciones en ambos extremos. Diseña el TAD ColaDoble con las operaciones usuales.

9.- Diseña un algoritmo que escriba todos los nodos de un árbol binario por niveles a partir del nivel 0. Por ejemplo, si tuviéramos el siguiente árbol

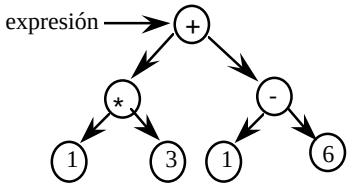


debería escribirse 4 2 5 1 3 6. Considera que "arbol" es de TipoArbol, estando éste declarado como en el ejercicio 13.

10.- Diseña una función que acepte el puntero a la raíz de un árbol binario y devuelva el número de nodos que contiene. Por ejemplo, para el árbol del problema anterior debería devolver 6.

11.- Diseña una función recursiva que compruebe si una cadena de caracteres (implementada con registros) representa una expresión en posfija válida.

12.- Un árbol binario puede utilizarse para almacenar una expresión algebraica, siendo las hojas los operandos y los demás nodos los correspondientes operadores.Por ejemplo:



representa a la expresión en notación prefija + * 1 3 - 1 6. Diseña un procedimiento recursivo que lleve una cadena de caracteres que representa una expresión en prefija a un árbol binario. Este estará implementado como se indica en la declaración de tipos del siguiente ejercicio.

13.- Muchos algoritmos sobre árboles binarios reflejan la naturaleza recursiva del árbol. Escribe procedimientos (o funciones) recursivas que realicen las siguientes tareas sobre árboles implementados como en clase, es decir:

```
TipoArbol = PUNTERO A Nodo
```

```
Nodo = REGISTRO
       Info:Z
       izq, der :TipoArbol
```

Nota: Se supone que los elementos no están repetidos.

a) Calcular la altura del árbol (el nivel mayor que contiene).
b) Encontrar el elemento mayor.
c) Calcular la suma de los elementos.
d) Encontrar un elemento dado.
e) Determinar el mayor nivel que está completo, es decir, que tiene el máximo número de nodos para ese nivel.
f) Borrar todos los nodos.
g) Insertar un elemento en el árbol.

14.- Una forma de codificar y decodificar mensajes con objeto de mantener su secreto consiste en usar el *método de la rejillas*. Una rejilla es una tarjeta perforada que, si se sitúa sobre una hoja de papel permite ver solo algunas porciones de la misma. En la codificación las letras del mensaje se escriben a traves de dichas perforaciones, y en la decodificación del mensaje se irán leyendo sucesivamente aquellas letras que sean visibles a traves de la rejilla.

Ejemplo:

```
  123456
1 XX XX
2 X XXXX
3 X X XX
4 XXXXXX
5  XX X
6 XX XXX
```

Hay perforaciones en las posiciones (1,3) (1,6) (2,2) (3,2) (3,4) (5,1) (5,4) (5,6) y (6,3).

Se podrán usar rejillas cuadradas con un máximo de seis filas y seis columnas.

Para codificar un mensaje cada letra del mismo se irá situando en el papel en la posición que quede libre en la rejilla, comenzando por la posición (1,1)..(1,6),(2,1)..(2,6) etc. Puede ocurrir que aun queden letras por codificar cuando se alcance la última posición libre de la rejilla. En éste caso se girará la rejilla 90º en algún sentido (de las agujas del reloj ó el opuesto), y se repetirá el proceso (la rejilla está diseñada para que no haya posibilidad de superponer dos letras). Mientras queden caracteres del mensaje por codificar, seguiremos girando la rejilla en el sentido seleccionado inicialmente y repitiendo el proceso.

Para decodificar un mensaje se usará la rejilla, superponiéndola sobre la hoja con el mensaje codificado, y se leerán los caracteres a través de los huecos de la rejilla, siguiendo el mismo mecanismo empleado al codificar. La rejilla se girará siempre cuatro veces, ya que después se encontrará en la posición inicial.

Se pide diseñar e implementar un TAD rejilla con las operaciones adecuadas para codificar y decodificar mensajes, que serán:

Algoritmo CrearRejilla(VAR rejilla: TipoRejilla; Numhuecos:N)
(* Crea una rejilla con Numhuecos, cuyas coordenadas serán leidas de teclado *)

Algoritmo Sentido(VAR rejilla:TipoRejilla;sentido:B)
(* Establece el sentido en el que se girará la rejilla
Si sentido = TRUE => El de las agujas del reloj
sentido = FALSE => El opuesto a las agujas del reloj. *)

Algoritmo Codificar(rejilla:TipoRejilla)
(* Lee un texto normal de teclado y muestra por pantalla el resultado de su codificación según la rejilla)

Algoritmo Decodificar(rejilla: TipoRejilla)
(* Lee un texto codificado de teclado y muestra por pantalla el resultado de descifrarlo según la rejilla indicada *)

15.- Un *mapa* es una función de un conjunto llamado dominio en otro conjunto llamado rango. Así que un mapa define una colección dinámica de vinculaciones del dominio en el rango. A cada vinculación le llamamos lazo. Un número arbitrario de lazos puede crearse, modificarse y destruirse durante la vida de un mapa. Si no hay lazos, consideramos que el mapa está vacío.

a) Diseña un TAD Mapa donde el dominio es un tipo enumerado (e1, e2, ..., en) cualquiera y el rango es cualquier tipo de datos. Las operaciones a implementar son:

```
Algoritmo CrearMapa(VAR mapa:Mapa)
(* crea un mapa vacío *)
Algoritmo AñadirLazo(d:Dominio;r:Rango;VAR mapa:Mapa)
(* Añade un lazo al mapa *)
Algoritmo EliminarLazo(d:Dominio;VAR mapa:Mapa)
(* Elimina un lazo del mapa *)
Algoritmo EstaAsociado(d:Dominio;mapa:Mapa;VAR lazo:B;VAR r:Rango)
(* pone lazo a TRUE y r al valor del Rango asociado a d si existe tal lazo; en otro caso pone lazo a FALSE *)
Algoritmo MapasIguales(mapa1,mapa2:Mapa):B
(* Determina si mapa1 y mapa2 son iguales *)
```

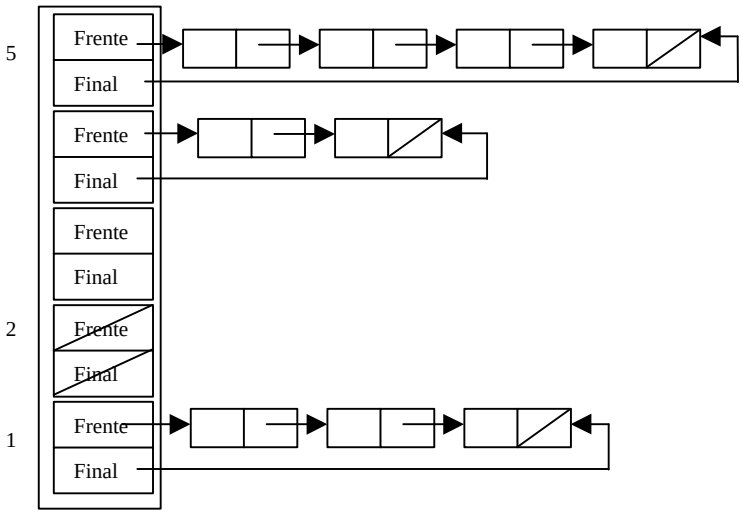
b) Si en el TAD Mapa Dominio = Color = (blanca, roja, negra) y Rango = CARDINAL, podemos implementar el TAD Urna. Una variable del tipo Urna representa una urna con un número cualquiera de bolas blancas, rojas y negras. Este número es el elemento del rango asociado al color correspondiente en el dominio. Por ejemplo, una urna con dos bolas negras y una blanca estaría representada por un mapa con los lazos (negra,2) y (blanca,1).

Utilizando los procedimientos ya implementados en el TAD mapa, diseña:

```
Algoritmo AñadirBola(c:Color;VAR urna:Urna)
(* Añade una bola de color c a la urna*)
Algoritmo SacarBola(VAR urna:Urna;VAR c:Color)
(* Extrae una bola de la urna. Supón que existe una función Aleatorio():Color
que devuelve un color de forma aleatoria al invocarla*)
```

16. Diseña el módulo de especificación y de implementación del TAD "ColaConPrioridad". Así mismo, diseña un módulo de utilización simple que utilice el TAD diseñado. Una cola con prioridad es aquella en la que se introducen valores con una prioridad especificada, y cuando se sacan valores de ella, se sacarán los valores introducidos con mayor prioridad. Para valores con igual prioridad el orden de salida es como una cola normal FIFO. Los elementos base de la cola serán números. Las operaciones que ofrece el TAD son:

```
CONST
    Min_Prioridad=1;
    Max_Prioridad=5;
TIPO
    ColaPrioridad= ???
ALGORITMO CrearCola(VAR cola: ColaPrioridad);
ALGORITMO DestruirCola(VAR cola:ColaPrioridad);
ALGORITMO MeterElem(VAR cola: ColaPrioridad; elem:Z; prio: N);
ALGORITMO SacarElemMaximaPrio(VAR cola:ColaPrioridad; VAR elem:Z);
ALGORITMO SacarElemMinimaPrio(VAR cola: ColaPrioridad; VAR elem:Z);
ALGORITMO ColaVacia(cola:ColaPrioridad):B
ALGORITMO Colallena(cola:ColaPrioridad):B
```



17. Definir a nivel de utilización de T.A:D. las siguientes funciones:

```
Algoritmo PilasIguales(p1,p2:TipoPila):B
(*Devuelve TRUE, si y sólo si, p1 y p2 son iguales*)
Algoritmo CopiarPila(p:TipoPila):TipoPila
(*Devuelve una copia de p*)
Algoritmo ConcatenarPilas(p1,p2: TipoPila):TipoPila
(*Devuelve una pila con los elementos de p2 sobre p1*)
```